[26] J. Zahorjan and C. McCann. "Processor Scheduling in Shared Memory Multiprocessors". In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 214–225, May 1990.

[14] S.-P. Lo and V.D. Gligor. "A Comparative Analysis of Multiprocessor Scheduling Algorithms". In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 205–222, September 1987.

[15] E. Markatos, M. Crovella, P. Das, C. Dubnicki, and T. LeBlanc. "The Performance of Scalable Barriers in the Presence of Multiprogramming". Technical Report 380, University of Rochester, Computer Science Department, May 1991.

[16] C. McCann, R. Vaswani, and J. Zahorjan. "A Dynamic Processor Allocation Policy for Multiprogrammed Shared Memory Multiprocessors". Technical Report 90-03-02, Department of of Computer Science and Engineering, University of Washington, March 1990 (Revised February 1991).

[17] J. K. Ousterhout. "Scheduling techniques for Concurrent Systems". In *1982 Distributed Computing Systems*, pages 22–30, October 18-22, 1982.

[18] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. "Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors". Technical Report 309, University of Rochester Computer Science Department, March 1989.

[19] M.L. Scott, T.J. LeBlanc, and B.D. Marsh. "Multi-Model Parallel Programming in Psyche". In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar 1990.

[20] M. S. Squillante. "Issues in Shared-Memory Multiprocessor Scheduling: A Performance Evaluation". Technical Report 90-10-04, Department of of Computer Science and Engineering, University of Washington, October 1990.

[21] M. S. Squillante and E. D. Lazowska. "Using Processor-Cache affinity Information in Shared-Memory Multiprocessor Scheduling". Technical Report 89-06-01, Department of of Computer Science and Engineering, University of Washington, February 1990.

[22] R.H. Thomas and W. Crowther. "The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors". In *Proc. 1987 International Conference on Parallel Processing*, pages 245–254, August 1987.

[23] A. Tucker and A. Gupta. "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors". In *Proceedings of the 12th Symposiun on Operating Systems Principles*, pages 159–166, Litchfield, AZ, Dec 1989.

[24] P.-C. Yew, N.-F. Tzeng, and D.H. Lawrie. "Distributing Hot-spot Addressing in Large-Scale Multiprocessors". *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

[25] J. Zahorjan, E. D. Lazowska, and D. L. Eager. "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Processors". Technical Report 89-07-03, Department of of Computer Science and Engineering, University of Washington, July 1989. to appear, *IEEE Transactions on Parallel and Distributed Systems*.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism". Technical Report 90-04-02, Department of of Computer Science and Engineering, University of Washington, Seattle, WA, October 1990.

[2] W.C. Athas and C.L. Seitz. "Multicomputers: Message-Passing Concurrent Computers". *IEEE Computer*, 21(8):9–24, August 1988.

[3] D. L. Black. "Scheduling Support for Concurrency and Parallelism in the Mach Operating System". *IEEE Computer*, 23(5):35–43, May 1990.

[4] D.G.Feitelson and Larry Rudolph. "Distributed Hierarchical Control for Parallel Processing". *IEEE Computer*, 23(5):65–77, May 1990.

[5] T. W. Doeppner Jr. "Threads: A System for the Support of Concurrent Programming". Technical report, Department of Computer Science, Brown University, 1987.

[6] D. L. Eager, E.D. Lazowska, and J. Zahorjan. "The limited Performance Benefits of Migrating Active Processes for Load Sharing". In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 63–72, May 24-27 1988.

[7] J. Edler, J. Lipkis, and E. Schonberg. "Process Management for Highly Parallel UNIX Systems". Technical Report Ultracomputer Note 136, Ultracomputer Research Laboratory, New York University, Apr 1988.

[8] R. Goldman and R. P. Gabriel. "Qlisp: Parallel Processing in Lisp". *IEEE Software*, 6(4):51–59, July 1989.

[9] A. Gupta, A. Tucker, and S. Urushibara. "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications". In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, May 1991.

[10] R. H. Halstead Jr. "Multilisp: A Language for Concurrent Symbolic Computation". *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[11] BBN Advanced Computers Inc. *Chrysalis Programmers Manual.* Cambridge, MA, Feb 1988. BBN multiprocessor.

[12] S. Leutenegger. *Issues in multiprogrammed multiprocessor scheduling.* PhD thesis, University of Wisconsin/Madison, August 1990. Technical Report 954.

[13] S.-P. Lo and V.D. Gligor. "Properties of Multiprocessor Scheduling Algorithms". In *Proceedings of the International Conference on Parallel Processing*, August 1987.

In summary, time-slicing introduces preemption, which can have enormous impact on a program, particularly programs that use barriers. Programs that don't use barriers, or synchronize infrequently are immune to the effects of time-slicing. Coscheduling has a cheap implementation and can remove the overhead due to preemption. Unfortunately, it has a built-in effectiveness of 80% or so, and performs poorly with unbalanced computations. Hardware partitions can be created fairly quickly, even when migration is required, and introduce minimal overhead due to context switching within a partition. Most important, hardware partitions allow an application to optimize its implementation for the percentage of the machine it is allocated. For this reason, hardware partitions are preferable to the other forms of multiprogramming.

6 Conclusions

There are many potential sources of overhead associated with multiprogramming, and the amount of overhead from any single source depends on the structure of applications. Given the efficient implementation of context switching in user space, and the relatively infrequent context switching required by the kernel, the overhead due to context switching is not a serious consideration. Preemption during synchronization is of considerable concern, and has serious consequences for applications that are time-sliced. This overhead can be avoided using coscheduling or hardware partitions.

There has not been much experimental evidence showing the relative importance of overhead due to processor-sharing, in particular with respect to coscheduling. Our results show that processor-sharing in the context of coscheduling performs significantly worse than dedicated hardware partitions wherein no processor-sharing occurs. In general, there are several reasons why this will be true: (1) coscheduling results in cache corruption, whereas hardware partitions do not; (2) there are fewer remote references and less contention when fewer processors are used; (3) there is less imbalance in the computation when the total amount of work is divided among fewer processors. These factors are significant enough to more than compensate for the costs of migration and the additional overhead of blocking synchronization (rather than busy-waiting) required within a hardware partition when the number of threads exceeds the number of available processors. Based on our experiences, we believe the best choice for multiprogramming on a large-scale machine is to use hardware partitions.

We see that, once again, the measured slowdown is much less than 2. In fact, in the case of coarse-grain threads with barrier synchronization, the slowdown is only 1.47, due to imbalance in the computation.

5.4 Comparison

A comparison between the three scheduling policies for each version of the gauss program is presented in figure 2. Each graph contains the execution time of the program for each policy in the presence of a background application.

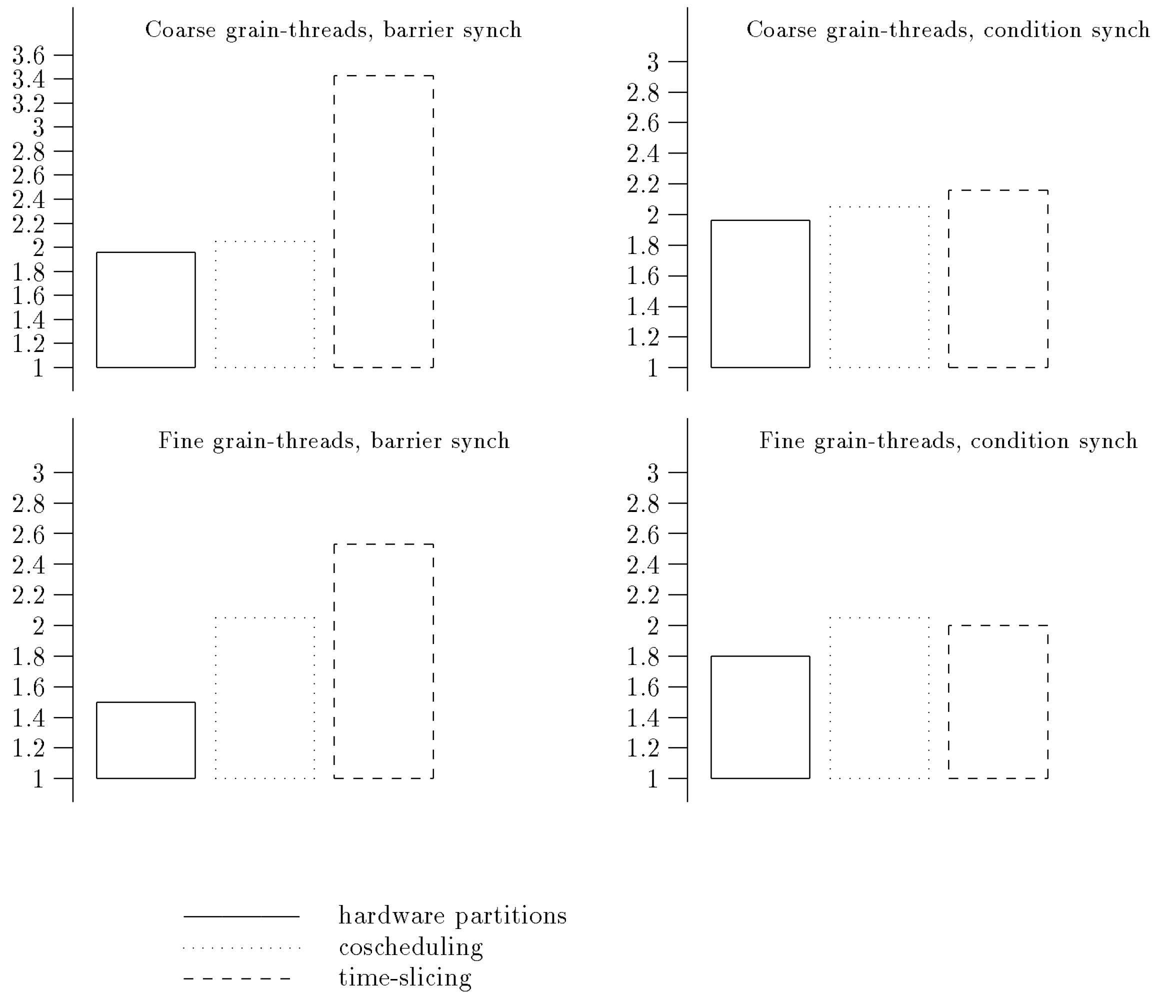

Figure 2: Relative Slowdown Introduced by a Multiprogramming Level of 2 for Different Scheduling Policies and Different Programming Models

It is evident from these graphs that in most cases time-slicing results in slowdown much higher than the expected factor of 2. Coscheduling's slowdown is uniformly slightly higher than 2. Hardware partitions incur slowdown which is better, sometimes substantially so, than 2.

shown in figure 1, the time required to execute the program with 16 threads on 8 processors is less than double the time used on 16 processors. These same results were observed for the program that uses condition synchronization. One reason for the better than expected performance on 8 processors is that there is significant contention for the pivot row on 16 processors, and much less on 8. Another reason is due to a slight imbalance in the computation, due to tail effects in the division of work in the matrix. In general, the applications can utilize 8 processors better than 16 processors because the speedup of the application is typically sublinear.

These experiments do not include the costs of migration. For those applications that cannot easily adapt the number of virtual processors in use, we must migrate threads when the hardware partition changes. To measure the effects of dynamic hardware partitions, we started the gauss program on 16 processors and then immediately introduced a background application. The arrival of the second application causes the operating system to divide the machine into two 8-processor partitions. The gauss application migrates 8 threads from the larger partition into the new smaller partition. To isolate the costs of migration, no computation was performed by gauss while holding 16 processors. The completion times of the application (in seconds) are shown below.

	standalone	with background application	slowdown
coarse-grain threads, barrier synch	18.7	36.9	1.97
coarse-grain threads, condition synch	18.1	35.5	1.96

These results show that even with the one-time cost of migration, and the recurring cost of multiplexing threads on a virtual processor, a hardware partition of 8 processors takes less than twice as long as a 16 processor partition. Clearly the lack of linear speedup in the application dominates the other sources of multiprogramming overhead. Therefore, the benefits of using hardware partitions can be expected to exceed the costs in most cases.

In the case of fine-grain threads, no migration or thread multiplexing is necessary. Instead, currently running threads (if any) are allowed to finish execution before removing a processor from a partition. The completion times of the application (in seconds) under dynamic hardware partitions for this case is presented below.

	standalone	with background application	slowdown
fine-grain threads, barrier synch	38	56	1.47
fine-grain threads, condition synch	25	45	1.8

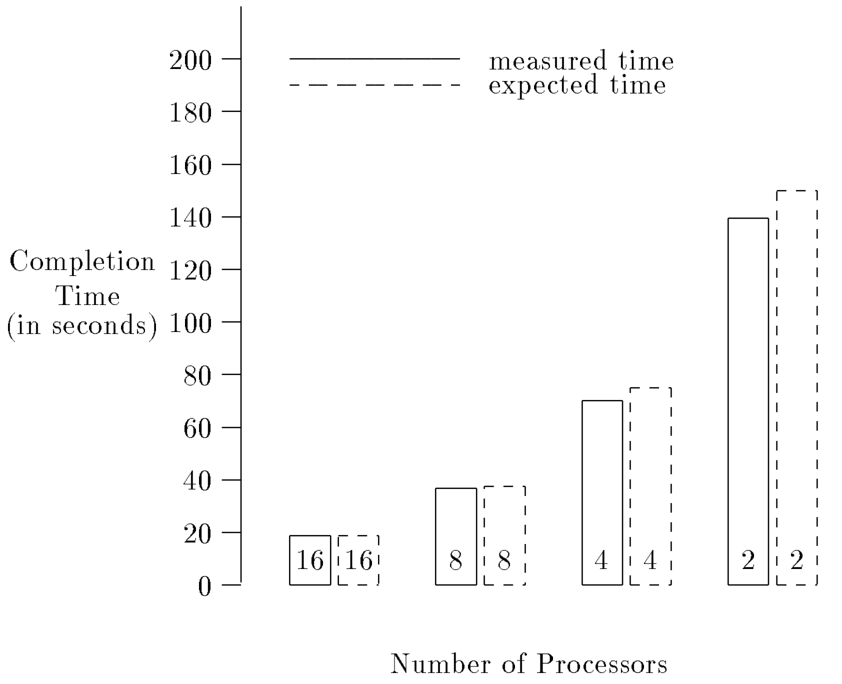

Figure 1: Gauss with 16 Threads and Barriers on Hardware Partitions

Next we considered whether unused slots in the processor-time matrix are of any use. That is, when an application is given extra cycles for one of its threads during a time when the other threads are not coscheduled, does this improve the running time of the application? To answer this question, we ran the gauss program with a background application that only used half the processors. This creates a timeslice during which the gauss program runs half its threads, followed by a timeslice in which all its threads run. The running time of the application in this case was 19.4 *sec,* as compared to 19.6 *sec* when sharing the machine with an application that uses all the processors during its quantum. This small difference suggests that unused slots do not contribute much to system throughput in the presence of synchronization. Ousterhout's simulations [17] show that coscheduling is typically 80% effective (measured in terms of the percent of processor time spent coscheduled); our result suggests that the effectiveness isn't improved by utilization of empty slots in the scheduling matrix.

5.3 Hardware Partitions

Our main concern in these experiments is the overhead introduced by multiplexing several threads on a single virtual processor and the cost of migration. To measure the overhead of multiplexing threads, we ran the gauss program for a 512×512 matrix with 16 coarse-grain threads and barrier synchronization on 16,8,4, and 2 processors. We observed the slowdown due to having fewer processors than threads; the results are shown in figure 1.

We would expect the execution time of the program on 8 processors to be at least double the time on 16 processors. The additional overhead of multiplexing threads should make the time on 8 processors even more than double the time on 16 processors. Nonetheless, as

We first note that thread creation time dominates in our fine-grain thread implementation. In addition, the communication costs associated with the fine-grain implementation are much higher than in the coarse-grain implementation. The barrier program experiences a slowdown of 2.5 in this case, compared with 3.4 earlier. The reason for this apparent improvement is that both programs have the same number of barriers (and hence the same opportunities for problems with preemption), but the duration of the fine-grain thread program is greater. As a result, there are three barriers per quantum in the coarse-grain program, and fewer than 1.5 barriers per quantum in the fine-grain program. It is the frequency of barriers in the coarse-grain program that produces the difference in slowdown.

Under condition synchronization both the execution time and the slowdown of the two versions are comparable. In the case of fine-grain threads, we see once again that condition synchronization is not frequent enough in our programs for preemption to significantly affect the execution time.

However, none of these versions of gauss synchronize extremely often because the finest grain implementation still must eliminate an element between synchronization points, and that takes several milliseconds. As a result, the barrier version only executes a barrier about once every 50 *ms*. Much worse cases of slowdown are possible with smaller matrices. In particular, a 256$\times$256 matrix problem slows down by a factor of 8 in the presence of one background application.

We used our implementation of odd-even sort to measure the effect of multiprogramming on programs that synchronize very frequently. On a dedicated machine, sorting an array of 512 elements takes 286 *ms*. The same program run with a job in the background takes 103 seconds, a slowdown factor of 366! The problem is caused by a combination of barriers, frequent synchronization (every 500 μs), and preemption. If we modify the implementation of barriers to yield the processor to another application rather than spin, giving up the rest of the quantum but receiving the next quantum sooner, we see a slowdown of 540; yielding the processor ensures that almost no barrier is ever completed within a quantum.

5.2 Coscheduling

In order to measure the costs of coscheduling, we ran the coarse-grained gauss programs with varying levels of processor sharing. The program used a 256$\times$256 matrix, and was run on four processors. Processor-sharing was introduced by injecting background applications that consisted of four coscheduled threads each. The results are shown below.

Number of Processes	Running Time (secs.)	Slowdown
1	9.8	1.00
2	19.6	2.01
3	29.5	3.01
4	39.3	4.01

This table shows that as the degree of multiprogramming rises, the execution time of a single application rises linearly, despite its need for synchronization. This behavior is in contrast to the case of time-slicing.

5 Experiments

We ran the four different versions of the gauss program on a 16 processor BBN Butterfly in order to show how a particular programming model performs under different multiprogramming policies, and to see if there is a single multiprogramming policy that behaves best in *most* cases.

For each scheduling policy, we ran the programs under two different scenarios: (1) under ideal conditions where only one application is in the system, and (2) under multiprogramming, with another applications in the background. Our multiprogramming experiments incorporate a compute-bound application in the background that consumes any cycles it is given. Our experimental results focus on the execution time of the parallel portion of an application; the serial portion of program loading and creation of virtual processors has not been included.

5.1 Time-Slicing

Our main concern in these experiments is the overhead introduced by preemption. We first ran the two implementations of gauss that use coarse-grain threads on a 512$\times$512 matrix under a time-slicing policy. Our results for 16 processors are:

	standalone	with background application	slowdown
coarse-grain threads, barrier synch	18.70	64.19	3.43
coarse-grain threads, condition synch	18.10	39.10	2.16

We would expect an application to take twice as long when the machine is shared with another application. In fact, a multiprogramming level of 2 introduces a slowdown of 3.43 on the program with barrier synchronization. Barrier programs are very sensitive to the effects of preemption, since the preemption of any one thread delays all threads.

The program based on condition synchronization was not adversely affected by multiprogramming. With a multiprogramming level of two, it experienced a slowdown factor of 2.16, very close to the expected. The reason that preemption does not distort this execution is that a thread does not depend on every other thread making progress during a short interval of time, as is true with barriers. Only the thread computing the next pivot row can delay other threads when preempted.

The results of our experiments for the fine-grain thread models are:

	standalone	with background application	slowdown
fine-grain threads, barrier synch	38.4	97.3	2.53
fine-grain threads, condition synch	24.35	49.1	2.01

of the thread is moved to a processor in the smaller partition, and the thread is placed on the ready queue of a virtual processor in that partition.

Several system calls were added to the kernel to support hardware partitions. The *register* and *unregister* system calls are used to request and release processor partitions. A parameter to the *register* call indicates whether or not the application (or programming model) is prepared to adapt the number of processes in use if a smaller partition is provided. This information is used by the kernel when processors are allocated. The kernel tries to assign each application a fair share of processors, but no more than requested. In addition, if an application is unable to dynamically adjust the number of virtual processors in use, and the kernel cannot provide all the processors requested, then an attempt is made to balance the work across a partition. In particular, if at most x processors are available to satisfy a request for y processors, then z processors are allocated, where z is the smallest integer less than or equal to x such that: $\lceil y/z \rceil = \lceil y/x \rceil$. In this way, the ready queues on the processors in a partition remain roughly in balance even when the application cannot adjust the number of virtual processors in use [citation omitted].

A *partition* system call returns the identity of the processors currently assigned to an application. When a thread package receives a signal from the kernel that the processor partition has changed, it uses this call to determine whether the executing processor is still in the partition. If not, it can migrate its thread to another processor in the partition.

A *migrate* system call allows for the migration of a memory object to a specified processor. The object is locked during migration, causing processors that access the object to fault, and wait until the end of the migration operation. Only thread state is ever migrated; code is replicated among the nodes in the partition, and isn't removed from a node until the associated program terminates. Also, migration can proceed in parallel on several nodes. For example, if a partition shrinks from 16 processors to 8, 8 threads can be migrated simultaneously to the remaining processors in the partition.

We currently migrate a minimum of one memory object (8K bytes) during migration. Each migration operation takes about 25 *ms* per memory object, which includes the cost of copying the memory object containing the state of the thread, unmapping the object in one address space and mapping it into another. In the worst case scenario we measured, we saw the cost of dynamically changing a system during execution from one 16-processor partition to two 8-processor partitions to be about 700 *ms*, where each process to be migrated contained 24K bytes of data.

Migration is fairly expensive in any system, and our implementation is no different. For simplicity, our implementation uses already existing kernel code to move memory objects; this code does not know that thread migration is taking place, and therefore does not realize that some operations, such as unmapping, are unnecessary. In addition, migration introduces enormous switch contention. Our implementation is sufficient for our experiments however, since (a) migration to a new partition only occurs when an application arrives or departs the system, a relatively infrequent occurrence, and (b) even relatively efficient migration is generally not helpful for short-term scheduling decisions [6].

the time-slicing system. However, if a coscheduled process blocks for communication which completes before its quantum is up, that same process will receive the rest of its quantum automatically, since the coscheduled priority range is higher than the background priority range.

Coscheduling requires that the processors in the multiprocessor be synchronized. All processors in our implementation use the same quantum duration, 100 *ms*, but we must also ensure that all processors begin a new quantum simultaneously. To implement this synchronization, we embed a tree barrier [24] in the clock handler of each processor. Our implementation uses a 4-way tree to combine the notification of arrivals to the barrier. Each processor is represented by a node in the tree. Each interior node waits for all its children to arrive at the barrier, and then informs its parent that it has also arrived at the barrier. When the root is so informed, it releases its children, and these in turn release their children, until all the leaves are released. Our combining tree is a 4-way tree because each processor can pack the information about its four children in one word, and with one comparison can determine if all four children have reached the barrier. The releasing tree is a binary tree.

The time required to synchronize 16 processors using this tree barrier is about 200 μs; the additional time required to make a scheduling decision using coscheduling is between 50 and 200 μs, depending on the number of applications. Without coscheduling the clock handler normally consumes about 200 μs each quantum, including the time to save state and make a scheduling decision. Our revised clock handler takes about 500 μs each quantum, or 0.5% of the quantum.

We also added two system calls to the kernel interface to support coscheduling. The first call reserves some number of slots in the coscheduling matrix for processes that are soon to be created. This call returns a handle for the application, so that when processes are created they can be added to the appropriate row of the coscheduling matrix. The second call creates a new process and places it in the coscheduling matrix in the given row.

4.3 Hardware Partitions

Our implementation of dynamic hardware partitions is similar to that described in [23], except that ours, being on a NUMA multiprocessor, also supports explicit migration. Our implementation requires cooperation between the operating system kernel and the library packages that implement the various programming models. The allocation of processors to applications is done in the kernel. Migration, which must occur when a partition grows or shrinks due to the departure or arrival of a new application, is implemented by the library package.

When a new application arrives or departs the system, the kernel notifies each application about changes in its hardware partition using a signal mechanism. If the partition shrinks so as to exclude a processor, the runtime library on that node may choose to either suspend the currently executing virtual processor and migrate the corresponding thread, or finish the thread and not allocate another to the virtual processor. The latter option is used in conjunction with the task queue model; explicit migration is used in all other cases. If a thread is migrated, the underlying virtual processor is suspended, the memory object

4.1 Time-Slicing

Our operating system on the Butterfly implements a straightforward extension of uniprocessor time-slicing. Users may create processes (represented by kernel processes) and bind them to physical processors. The kernel time-slices among the processes on a processor. Processes are never migrated.

Each processor has a ready queue that is sorted by process priority. Within a priority level, processes are served in a round-robin fashion. Each process gets a fair share of the processor; as in Unix, a user with many processes can get more cycles than a user with few processes.

4.2 Coscheduling Implementation

We implemented coscheduling using an adaptation of Ousterhout's matrix algorithm [17] and the time-slicing kernel described above. We chose this approach for simplicity, and to address two specific problems that arise when coscheduling is used in a system with priorities and blocking processes.

Kernel processes block for a variety of reasons, including while waiting for I/O or for communication with another kernel process. When a kernel process blocks, the alternate selection mechanism used to fill unused slots in the scheduling matrix could be invoked to select another user-level process to execute the remainder of the quantum. However, if the blocked process is unblocked during the same coscheduled quantum, we would like to return the processor to that process, without overly complicating the scheduler.

We would also like to maintain the system of priorities used to implement kernel processes. Priorities are used in many operating systems to implement a fair allocation of resources and to ensure that critical operations, such as I/O, proceed immediately. We need to integrate priority scheduling and coscheduling in the same implementation.

Previous algorithms for coscheduling [17, 4] did not consider the effects of priorities or blocking kernel processes. For example, in Ousterhout's matrix algorithm, there is no efficient way to implement priorities without scanning the entire matrix on each scheduling decision. Similarly, there is no notion of whether a process is runnable or not, so the concept of yielding the processor to a runnable process is difficult to implement. For this reason, we adapted the priority-queue implementation of time-slicing to include coscheduling.

The priority range implemented in the kernel is separated into *immediate*, *coscheduled*, and *background* ranges. The highest priority processes are in the immediate range, and are assigned to kernel processes that implement I/O handlers. At any one time, the coscheduled priority range is occupied by at most one process on each processor. The background range implements a round-robin set of runnable applications.

At each quantum boundary, a single process on each processor is elevated to the coscheduled range, while the previously coscheduled process is demoted to the background range. In addition, a matrix is maintained of all jobs in the system, just as in traditional coscheduling. This matrix does not determine which processes run however, it only directs the promotion of processes to the coscheduled priority range. All processes occupy some place in the matrix, so no process will starve. Both priorities and process blocking are handled as in

We implemented four different versions of gauss, representing different parallelizations of row elimination. The first implementation uses stateless threads and condition (neighbor) synchronization. The main program distributes the problem matrix among the memories of the machine, creates one virtual processor per physical processor, and then creates a global queue of threads, which are assigned to virtual processors. Each thread eliminates some number of entries in the matrix. Before eliminating an entry, the thread checks to see if the condition flags associated with the pivot row and the entry are set. If so, the two rows are copied into the local memory, the computation is performed, and the result is copied back into the original matrix. When a thread terminates, a virtual processor is assigned a new thread. This program is analogous to the task queue model.

The second implementation is similar to the first, except that it uses barrier synchronization. The threads that eliminate entries in a single column of the matrix synchronize using a barrier upon completion. Then, a new set of threads is generated for the next column. The copy costs are the same in both versions.

The third implementation uses coarse-grain threads and condition synchronization. The main program creates one virtual processor per physical processor, and assigns a singe thread to each virtual processor. The rows of the matrix are distributed among the threads in a round-robin fashion. Each thread eliminates all the entries in several rows. There is less synchronization than in the earlier version based on condition synchronization, since only synchronization with the pivot row is necessary. In addition, there is unlikely to be much spinning, since the pivot row is the first computation performed in each phase of execution. Most important, there are many fewer row copy operations performed with coarse-grain threads; $O(N^2)$ instead of $O(N^3)$.

The final implementation of gauss uses coarse-grain threads with barriers. Each thread eliminates some elements in a single column of the matrix, synchronizes with the other threads using a barrier, and then proceeds to the next column.

The sort program creates one virtual processor per physical processor, and assigns one thread to each virtual processor. The array to be sorted is divided among the virtual processors in the application. Each thread performs N/P/2 comparisons in each phase, and then synchronizes with the other threads using a barrier. The length of a phase is a few milliseconds for an array of several thousand elements on 16 nodes.

In the following section we describe the results of our experiments that use these programs to compare three multiprogramming policies.

4 Multiprogramming Implementation

We implemented our multiprogramming experiments on a BBN Butterfly multiprocessor. We modified an existing operating system for time-slicing among applications to implement coscheduling and hardware partitions.

Coarse-grain threads do not share these characteristics. Once a coarse-grain thread creates its state on a processor, it cannot be easily moved to another one. If such a thread is executing when preemption occurs, there are few options to avoid the overhead associated with preemption in the presence of synchronization. If a repartition of the physical machine is required, the cost of migration will be high.

3.3 Synchronization

One important source of overhead in multiprogrammed systems is due to preemption in the presence of synchronization. There are several different kinds of synchronization however, and each type of synchronization has several implementations. The overhead introduced by preemption varies both with the type of synchronization used and the implementation. For example, peemption of programs that use spin locks could seriously affect performance, whereas preemption might not significantly affect programs that use blocking semaphores. Similarly, preemption has less impact on programs that use centralized barriers instead of tree barriers [15]; there is only one synchronization point in a centralized implementation, whereas there are two or more synchronization points in a tree barrier implementation.

The amount of overhead due to synchronization also depends greatly on the frequency of synchronization. In the worst case, an application may require a scheduling decision at every synchronization point in the program because of multiprogramming.

There are three classes of synchronization: mutual exclusion, condition synchronization, and barriers. We do not consider mutual exclusion in our experiments; it has been shown that for small critical sections that are not already a bottleneck, preemption does not impose undue overhead [25]. Nearly all programs that use spin locks have small critical sections with low utilization. In addition, blocking, or spinning for a short interval and then blocking if necessary, reduces the cycles lost to spinning while waiting for a preempted thread, and performs as well as pure spinning in the absence of preemption.

Our primitive for condition synchronization uses 2-phase blocking [13], where a process spins for a short while and then blocks if the condition is not satisfied, yielding the processor to another thread in the same application. This primitive has most of the performance advantages of spinning, but suffers much less in the presence of preemption. We use a tree barrier implementation that yields the processor to another thread on the same virtual processor if it exists, and spins otherwise. Although both of these primitives work well with an arbitrary number of threads, they can waste cycles spinning when applications share a processor.

3.4 Example Programs

Our experiments were performed using two different applications: gaussian elimination and sorting. We chose gaussian elimination because it has several different decompositions that allow us to measure the impact of the programming model and multiprogramming on the same basic application. We chose odd-even sort because it uses very frequent synchronization.

processor.[2] No two virtual processors from the same program need share a processor, although processors might have to be shared with other applications. Once execution begins, these programs cannot easily adapt to fewer virtual processors, since to do so may require migration of threads from one virtual processor to another, and multiplexing of threads on a single virtual processor.

Some applications can adapt the number of virtual processors in use dynamically during execution, without requiring that we multiplex threads on a single virtual processor. For example, the task queue model used in Multilisp [10], Qlisp [8], and BBN's Uniform System [22], and employed by applications that use packages such as Brown's thread package [5], does not depend on the number of available virtual processors. Parallelism in an application is represented by tasks in the queue, which can be mapped to any number of virtual processors. The number of virtual processors can vary during execution, so long as no virtual processor is halted while executing a task.

Each class of application may exist simultaneously within a multiprogrammed multiprocessor, and will be affected to different degrees by the multiprogramming policy. Applications that can easily adapt the number of virtual processors in use may prefer dedicated processors over processor-sharing, so as to remove all multiprogramming overhead. Applications that cannot easily adapt the number of virtual processors in use, and are forced to multiplex threads on a single virtual processor, may prefer coscheduling or time-slicing over dedicated processors, so as to avoid excessive thread context switching that might arise in a small partition. The total overhead introduced by multiprogramming depends on the extent to which the affected applications are well-matched to the policy in place.

3.2 Threads

We assume that virtual processors are scheduled by the system and are subject to multiprogramming; threads are created and managed in user space by a thread package, and therefore are not directly under control of the operating system. Nonetheless, the number and type of thread used in an application can have an impact on multiprogramming. In particular, the costs of moving threads from one partition to another depend in part on the type of thread used in an application. In addition, the lifetime of a thread determines how easily we can adapt the number of virtual processors in use.

Fine-grain threads, which are typically used to represent the natural grain of parallel activity in an application, are short-lived and relatively stateless. Coarse-grain threads, on the other hand, execute longer and build up state in the cache and the local memory. As a result, a program based on fine-grain threads offers more opportunities for adaptation in a multiprogrammed environment.

Fine-grain threads come and go frequently, so any virtual processor that must give up its physical processor (either due to preemption or partition) can wait for a thread to terminate before doing so. The existence of such a clean point greatly simplifies multiprogramming with dynamic hardware partitions [23], and can be used to minimize the overhead caused by preemption during synchronization.

[2]Since virtual processors correspond to kernel processes, there is nontrivial overhead associated with their creation, and little reason to create more than one per physical processor.

3 User-Level Programming Models

There are many different parallel programming models in use today. Our goal is to explore the interactions between the programming model employed by an application and the multiprogramming policy implemented by the operating system. Rather than attempt to cover all parallel programming languages and packages, we focus on a set of general attributes shared by many models currently in use.

We assume an application consists of a relatively large number of threads that are mapped to a relatively small number of virtual processors. Threads represent potential concurrency, while virtual processors are intended to represent true parallelism. An application has control over the number of threads used. We assume that an application creates at most one virtual processor per physical processor, but a single virtual processor may execute many threads concurrently[1]. Depending on the mapping between threads and virtual processors, threads may each have their own address space (heavyweight processes), share a single address space (lightweight processes or threads), or operate within overlapping address spaces. We assume that a context switch between threads in the same application is implemented in user space by the runtime environment of the programming model, and therefore is reasonably efficient. Context switching between virtual processors is implemented by the kernel.

The attributes of an application most directly related to multiprogramming are the ability of virtual processors to adapt in number, the number and type of threads used, and the frequency and type of synchronization.

3.1 Virtual Processors

Virtual processors are scheduled by the kernel. They may correspond one-to-one with physical processors, as is required in Tucker and Gupta's multiprogramming solution [23]. Alternatively, there be many more virtual processors than physical processors, as in coscheduling and time-slicing for example, wherein virtual processors from different applications share a physical processor. Some form of processor-sharing is required anytime the number of virtual processors in the system exceeds the number of physical processors.

In some applications the number of virtual processors required for execution is fixed at the time the program is written. Static parallelism is normally used to represent the functional parallelism in a program. Typically, the granularity of functional parallelism is high, and the number of virtual processors required to represent functional parallelism in a program is small. These programs are not included in our comparison.

Most applications are capable of adapting to the multiprocessor environment at the time the program begins execution. In this case the operating system can tell the application how many processors are available, and the program can create one virtual processor per physical

[1]We could multiplex several virtual processors from a single application on one physical processor, but to do so would introduce unnecessary context switching in the kernel. Because of this assumption, our measurements of the overhead of multiprogramming are conservative.

Tucker and Gupta [23] proposed a combination of dedicated processor scheduling and a programming model that dynamically adjusts the number of processes in an application to equal the number of processors in the partition. Their model, which assumes the use of fine-grain threads in the application, can suspend a kernel process between the execution of two threads. Their experiments show that having one process per processor results in significant performance improvement when compared to a time-slicing policy. Subsequent work by Gupta *et al.*[9] investigated the effects of different scheduling policies and synchronization primitives on an UMA multiprocessor using simulation. They showed that in the presence of multiprogramming blocking primitives always outperform spinning primitives. They also showed that coscheduling and hardware partition policies are better than traditional round-robin prioritized polices due to their high cache hit ratio and low synchronization overhead. Moreover, hardware partitions along with process control [23] typically outperform coscheduling because hardware partitions usually achieve higher processor utilization.

Unfortunately, not all applications can easily adjust the number of running processes on demand. Some programming models encourage applications to create a static number of processes, so as to avoid unnecessary process creation, destruction, and context switching. Others use coarse-grain threads of control, which reduce the opportunities for dynamic adjustment. Although the programming model used by Tucker and Gupta is widely used, their work does not characterize the effects of multiprogramming on applications that do not adhere to the model.

Zahorjan and McCann [26] simulated the performance of hardware partitions with a workload containing programs that change their parallelism frequently. They concluded that a dynamic hardware partition policy is the best choice, since such a policy can reallocate unused processors immediately. Subsequent experimental work by McCann, Vaswani, and Zahorjan [16] on a Sequent Symmetry confirms this conclusion. The same argument may not be valid for a NUMA multiprocessor however, since processor reallocation may be too expensive to perform every time an application changes the amount of parallelism it employs. In addition, applications with a fixed amount of parallelism that synchronize very frequently may prefer coscheduling over hardware partitions, since a small hardware partition may force them to incur context switch overhead on every synchronization operation.

2.4 Research Goals

Many open questions remain regarding multiprogramming on multiprocessors. Each technique described above is known to address some source of overhead, but no technique addresses all sources of overhead for all programs. Time-slicing with notification of preemption doesn't address overhead due to frequent synchronization. Coscheduling doesn't consider the penalty incurred by processor-sharing. The costs associated with dynamic hardware partitions have not been fully explored, including the costs of multiprogramming within a partition and the costs of processor reallocation on a NUMA multiprocessor. Our goal is to explore these costs for a range of programming styles using an actual implementation of each approach.

Psyche either notifies a process asynchronously when preemption is imminent [18], or sets a flag that can be examined by a process before it enters a critical section [19]. Washington's activations [1] pass the state of a process that is preempted in a critical section to another processor under control of the same application.

Many operating systems support a single centralized ready queue from which processes are dispatched according to their priority. This approach is very popular on small-scale UMA machines. However, Squillante and Lazowska [20, 21] have shown that by ignoring the affinity that may have been created between a process and a processor, a centralized ready queue can introduce a performance penalty of 99%, with 69% due to cache reload and 30% due to increased bus traffic and contention. Their suggested solution, local ready queues for short-term scheduling, and a global queue for longer-term load balancing, does not take other sources of overhead into account however.

2.2 Coscheduling

Coscheduling was originally proposed by Ousterhout [17] to address the overhead related to synchronization. With coscheduling, the processes in an application all run at the same time. There are two important advantages to coscheduling: no process is forced to wait for another that has been preempted and processes may communicate without an intervening context switch. There are also disadvantages, however. If there are several applications in the system, the machine must cycle through each of them, during which time the caches can be expected to lose any contents related to an earlier execution [23]. Second, utilization may suffer if applications have a variable amount of parallelism, or if processes cannot be evenly assigned to time-slices of the machine.

Leutenegger [12] used simulation to evaluate the performance of several different policies, including coscheduling. He showed that for programs with very frequent communication, a policy that schedules the processes of an application to run at the same time perform significantly better than schedulers that do not have this property. This study did not consider cache or memory affinity, or programs with a variable amount of parallelism.

2.3 Hardware Partitions

When hardware partitions are used, no two applications share a processor. A set of processors may be dedicated to an application for a relatively long fixed interval [3] or for the entire duration of the application [11, 2]. Within its own hardware partition, each application may choose to allocate one process per processor, thereby avoiding entirely the overhead attributed to multiprogramming. However, to ensure fairness and to efficiently utilize the processors, the number of processors assigned to an application might have to change when another application arrives or departs [23], or when the degree of parallelism changes within an application [26, 16]. Unless an application can easily adjust the number of processes it employs during execution, several processes from the same application may have to share a processor, introducing context switching and other related sources of overhead.

number of processes per application, the amount of state associated with a process, and the frequency and type of synchronization. Due to the complexity of the problem, many of the tradeoffs inherent in multiprogramming have been examined only in the context of specific architectures and programming models, and in many cases using simulations. There has been little experimental comparison of the various solutions in the presence of applications with varying degrees of parallelism and synchronization.

In this paper we experimentally compare the performance of three different multiprogramming schemes: time-slicing, coscheduling, and dynamic hardware partitions. We modified an existing operating system to implement the three different schemes, and then implemented several applications that vary in degree of parallelism, and the frequency and type of synchronization. Our experiments were performed on a NUMA multiprocessor without caches, but most of our results apply equally well to both UMA architectures and multicomputers such as the Hypercube. Our results show that in most cases coscheduling is preferable to time-slicing. Our results also show that although there are cases where coscheduling is beneficial, dynamic hardware partitions do no worse, and will often do better. We conclude that under most circumstances, hardware partitioning is the best strategy for multiprogramming a multiprocessor, no matter how much parallelism applications employ or how frequently synchronization occurs.

2 Multiprogramming Techniques

Many different multiprogramming schemes have been proposed or implemented on multiprocessors, but most are derived from one of three basic approaches: unsynchronized time-sharing (time-slicing), synchronized time-sharing (coscheduling), and space-sharing (hardware partitions).

2.1 Time-Slicing

Time-slicing a multiprocessor is a straightforward adaptation of uniprocessor time-slicing, and is frequently employed in operating systems derived from uniprocessor systems. Each application (process) is given a fair share of the machine (processor). At the end of a quantum, a processor selects the next process to run from the ready queue, which may or may not be shared with other processors. There is no coordination among the processes of an application with respect to when they run or even where they run. In particular, processes within an application might not all be assigned to different processors, and there is no guarantee that any two processes will ever run simultaneously.

Although a useful technique for balancing load across the applications in a system, multiprocessor time-sharing can suffer severe performance penalties. Since there is no guarantee that an application's processes will run at the same time, processes may be blocked while waiting for a preempted process or may be required to context switch after every synchronization operation. Several studies have shown that this effect can lead to severe performance degradation [12, 13, 14, 23].

Several systems incorporate a special mechanism to avoid preemption while in a critical section. SymUnix allows a process to delay preemption until it leaves a critical section [7].

1 Introduction

Multiprocessors are an expensive resource that must be shared. Sharing requires a delicate balance between fairness and resource utilization that is usually achieved through multiprogramming. In order to be effective however, multiprogramming overhead must be minimized. There are several potential sources of overhead in a multiprogrammed multiprocessor environment, and each can significantly affect system performance.

Context switch overhead is introduced when processes share a processor. Even though many multiprocessor thread packages provide a user-level context switch that does not require kernel intervention, there may still be a need for several kernel processes to share a processor. The frequency of context switching through the kernel, and therefore the amount of overhead, depends on the quantum size (when processes share a processor using time-slicing) and the frequency of communication or synchronization (which may cause one process to block and another to run).

A second source of overhead is due to preemption in multiprogrammed systems that use time-slicing. If a process is preempted while inside a critical section or while computing some condition on which other processes depend, then processes may waste their quantum waiting for the preempted process to run. If processes spin while waiting, then many processor cycles are wasted by spinning. Even if processes block during synchronization, they must context switch and lose the remainder of their quantum.

A third source of overhead is the cost of cache reload, remote memory references, and migration incurred when a process is moved from one processor to another. During execution a process builds state on a processor, either in the cache of a uniform memory access (UMA) multiprocessor, or in the local memory of a nonuniform memory access (NUMA) machine. If the process is then assigned to another processor, it must reload the cache on an UMA, and issue remote references or migrate the contents of memory on a NUMA. Even if a process is not moved to a different processor, other applications can corrupt the cache while it is preempted, forcing a cache reload. If the cache miss penalty is high, the associated overhead can have a serious impact on performance.

A fourth source of overhead, and one that has not received much attention, arises whenever parallel applications share processors. Every parallel program strikes a balance between the benefits of parallel execution and the overhead of parallelism *in the absence of processor-sharing*. When a multiprogramming policy causes applications to share a processor, the overhead of parallelism remains, but the effective speed of the processors appears to decrease. As a result, the balance between effective parallelism and overhead embodied in a program can be upset by the multiprogramming policy, which results in inefficient execution. In particular, an application with nonlinear speedup may prefer a small number of dedicated processors to a larger number of shared processors, even when the processor-time product is the same in both cases.

It is extremely difficult to find a single multiprogramming policy that can maximize processor utilization, ensure fairness, and simultaneously address all of these sources of overhead. The costs associated with a particular policy depend on the underlying architecture: the cache miss penalty, the remote access penalty, and the cost of migration. The performance implications of a policy depend on the characteristics of the applications: the

Multiprogramming on Multiprocessors

Mark Crovella Prakash Das Czarek Dubnicki
Thomas LeBlanc Evangelos Markatos

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 385

February 1991
(revised May 1991)

Abstract

Several studies have shown that applications may suffer significant performance degradation unless the scheduling policy minimizes the overhead due to multiprogramming. This overhead includes context switching among applications, waiting time incurred by one process due to the preemption of another, and various migration costs associated with moving a process from one processor to another. Many different multiprogramming solutions have been proposed, but each has limited applicability or fails to address an important source of overhead. In addition, there has been little experimental comparison of the various solutions in the presence of applications with varying degrees of parallelism and synchronization.

In this paper we explore the tradeoffs between different approaches to multiprogramming a multiprocessor. We modified an existing operating system to implement three different multiprogramming options: time-slicing, coscheduling, and dynamic hardware partitions. Using these three options, we implemented applications that vary in the degree of parallelism, and the frequency and type of synchronization. We show that in most cases coscheduling is preferable to time-slicing. We also show that although there are cases where coscheduling is beneficial, dynamic hardware partitions do no worse, and will often do better. We conclude that under most circumstances, hardware partitioning is the best strategy for multiprogramming a multiprocessor, no matter how much parallelism applications employ or how frequently synchronization occurs.

This work was supported in part by NSF grant CCR-9005633 and an NSF Institutional Infrastructure Program grant CDA-8822724.